# AN643

# Adaptive Differential Pulse Code Modulation Using PIC® Microcontrollers

*Author:* *Rodger Richey*
*Microchip Technology Inc.*

## INTRODUCTION

In the past, adding speech recording and playback capability to a product meant using a digital signal processor or a specialized audio chip. Now, using a simplified Adaptive Differential Pulse Code Modulation (ADPCM) algorithm, these audio capabilities can be added to any PIC® microcontroller or DSC digital signal controller. This application note will cover the ADPCM compression and decompression algorithms, and an application using a PIC18F67J10 microcontroller.

## DEFINITION OF TERMS

**step size** – value of the step used for quantization of analog signals and inverse quantization of a number of steps.

**quantization** – the digital form of an analog input signal is represented by a finite number of steps.

**adaptive quantization** – the step size of a quantizer is dramatically changed with time in order to adapt to a changing input signal.

**inverse quantizer** – a finite number of steps is converted into a digital representation of an analog signal.

**adaptive predictor** – a time varying algorithm that calculates an estimate of the input signal from the quantized difference output.

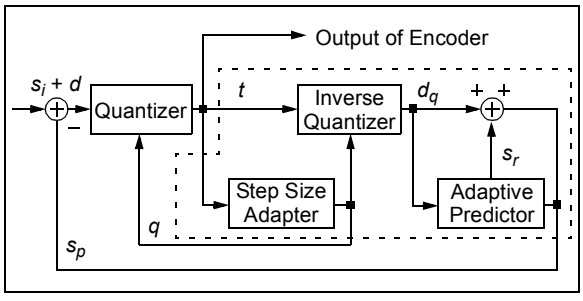
## THEORY OF OPERATION

The ADPCM algorithm takes advantage of the high correlation between consecutive speech samples, which enables future sample values to be predicted. Instead of encoding the speech sample, ADPCM encodes the difference between a predicted sample and the speech sample. This method provides more efficient compression with a reduction in the number of bits per sample, yet preserves the overall quality of the speech signal. The implementation of the ADPCM algorithm provided in this application note is based on the now defunct Interactive Multimedia Association's (IMA) Recommended Practices for Enhancing Digital Audio Compatibility in Multimedia Systems revision 3.00. The ITU (formerly CCITT) G.721 ADPCM algorithm is well known and has been implemented on many digital signal processors, such as the TMS320™ DSP family of devices from Texas Instruments™ and the ADSP-2100 family of devices from Analog Devices. ITU G.721 uses floating-point arithmetic and logarithmic functions, which are not easily implemented in the 8-bit microcontroller world. The IMA Reference Algorithm significantly reduces the mathematical complexity of ITU G.721 by simplifying many of the operations and using table look ups where appropriate.

## COMPRESSION

The input, $s_i$, to the encoder routine must be 16-bit two's complement speech data. The range of allowable values for $s_i$ is 32767 to -32768. Figure 1 shows a block diagram for ADPCM compression and **Appendix B: "Generic `ADPCMEncoder()` Function"** lists the generic C code for the `ADPCMEncoder ()` function. The predicted sample, $s_p$, and the quantizer step size index ($q$) are saved in a structure for the next iteration of the encoder. Initially, the quantizer step size index and the predicted sample ($s_p$) are set to zero. The encoder function takes a 16-bit two's complement speech sample and returns an 8-bit number containing the 4-bit sign-magnitude ADPCM code *(t*).

**FIGURE 1: ADPCM ENCODER BLOCK DIAGRAM**




The predicted sample, $s_p$, is subtracted from the linear input sample, $s_i$, to produce a difference, *d*. Adaptive quantization is performed on the difference, resulting in the 4-bit ADPCM value, *t*. The encoder and decoder both update their internal variables based on this ADPCM value. A full decoder is actually embedded within the encoder. This ensures that the encoder and decoder are synchronized without the need to send any additional data. The embedded decoder is shown

within the dotted lines of Figure 1. The embedded decoder uses the ADPCM value (*t*) to update the inverse quantizer, which produces a dequantized version, $d_q$, of the difference, *d*. The implementation of the ADPCM algorithm presented in this application note uses a fixed predictor, instead of an adaptive predictor, which reduces the amount of data memory and instruction cycles required.

The adaptive predictor of ITU G.721 adjusts according to the value of each input sample, using a weighted average of the last six dequantized difference values and the last two predicted values. So, at this point, the dequantized difference, $d_q$, is added to the predicted sample, $s_p$, to produce a new predicted sample, $s_r$, as shown in Figure 1. Finally, the new predicted sample, $s_r$, is saved into $s_p$ to be used for the next iteration.

Table 1 provides a step-by-step description of the `ADPCMEncoder()` function, which is listed in **Appendix B: "Generic `ADPCMEncoder()` Function"**.

**TABLE 1: `ADPCMEncoder()` STEP-BY-STEP FUNCTIONS**

| | | |
|---|---|---|
| 1. | `ADPCMEncoder` takes a 16-bit signed number (speech sample, 32767 to -32768) and returns an 8-bit number containing the 4-bit ADPCM code (0-15). | `char ADPCMEncoder(long signed sample)` |
| 2. | Restore the previous values of predicted sample ($s_p$) and the quantizer step size index. | `predsample = state.prevsample;`<br>`index = state.previndex;` |
| 3. | Find the quantizer step size (*q*) from a table look up using the quantizer step size index. | `step = StepSizeTable[index];` |
| 4. | Compute the difference (*d*) between the actual sample ($s_i$) and the predicted sample ($s_p$). | `diff = sample - predsample;` |
| 5. | Set the sign bit of the ADPCM code (*t*), if necessary, and find the absolute value of difference (*d*). | `if(diff >= 0)`<br>`  code = 0;`<br>`else`<br>`{`<br>`  code = 8;`<br>`  diff = - diff;`<br>`}` |
| 6. | Save quantizer step size (*q*) in a temporary variable. | `tempstep = step;` |
| 7. | Quantize the difference (*d*) into the ADPCM code (*t*) using the quantizer step size (*q*). | `if(diff >= tempstep)`<br>`{`<br>`  code \|= 4;`<br>`  diff -= tempstep;`<br>`}`<br>`tempstep >>= 1;`<br>`if(diff >= tempstep)`<br>`{`<br>`  code \|= 2;`<br>`  diff -= tempstep;`<br>`}`<br>`tempstep >>= 1;`<br>`if(diff >= tempstep)`<br>`  code \|= 1;` |
| 8. | Inverse quantize the ADPCM code (*t*) into a predicted difference ($d_q$) using the quantizer step size (*q*). | `diffq = step >> 3;`<br>`if(code & 4)`<br>`  diffq += step;`<br>`if(code & 2)`<br>`  diffq += step >> 1;`<br>`if(code & 1)`<br>`  diffq += step >> 2;` |

**TABLE 1: `ADPCMEncoder()` STEP-BY-STEP FUNCTIONS (CONTINUED)**

| | |
|---|---|
| 9. Fixed predictor computes new predicted sample ($s_r$) by adding the old predicted sample ($s_p$) to the predicted difference ($d_q$). | `if(code & 8)`<br>`  predsample -= diffq;`<br>`else`<br>`  predsample += diffq;` |
| 10. Check for overflow of the new predicted sample ($s_r$). $s_r$, which is a signed 16-bit sample, must be in the range of 32767 to -32768. | `if(predsample > 32767)`<br>`  predsample = 32767;`<br>`else if(predsample < -32768)`<br>`  predsample = -32768;` |
| 11. Find the new quantizer step size index (q) by adding the previous index and a table look up using the ADPCM code ($t$). | `index += IndexTable[code];` |
| 12. Check for overflow of the new quantizer step size index. | `if(index < 0)`<br>`  index = 0;`<br>`if(index > 88)`<br>`  index = 88;` |
| 13. Save the new predicted sample ($s_r$) and quantizer step size index for next iteration. | `state.prevsample = predsample;`<br>`state.previndex = index;` |
| 14. Return the ADPCM code ($t$). | `return (code & 0x0f);` |

This function requires five 16-bit variables and two 8-bit variables. Some optimizations can be made to the code, which is listed in **Appendix B: "Generic `ADPCMEncoder()` Function"**, such as combining steps 7 and 8.

## DECOMPRESSION

The input into the decoder, $t$, must be an 8-bit number containing the 4-bit ADPCM data in sign-magnitude format. The range of allowable values for $t$ is 0 to 15, where 7 = 0x07 and -7 = 0x0F. Figure 2 shows a block diagram for ADPCM decompression and **Appendix C: "Generic `ADPCMDecoder()` Function"** lists the generic C code for the `ADPCMDecoder()` function.

The predicted sample, $s_p$, and the quantizer step size index are saved in a structure for the next iteration of the decoder. Initially, the quantizer step size index and the predicted sample ($s_p$) are set to zero. This function takes a 4-bit sign-magnitude ADPCM code and returns the 16-bit two's complement speech sample.

This decoder is the same as the one used in the encoder routine. It uses the ADPCM value to update the inverse quantizer, which produces a difference, $d_q$. The difference, $d_q$, is added to the predicted sample, $s_p$, to produce the output sample, $s_r$. The output sample, $s_r$, is then saved into the predicted sample, $s_p$, for the next iteration of the decoder.

**FIGURE 2: ADPCM DECODER BLOCK DIAGRAM**

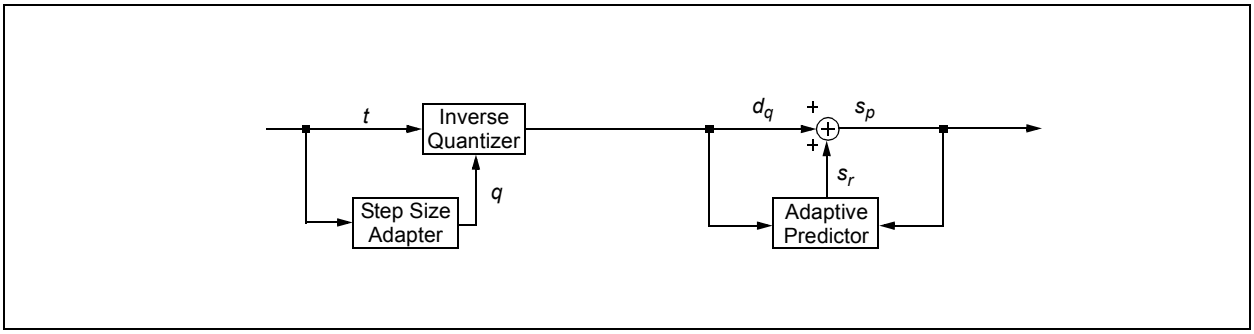


Table 2 provides a step-by-step description of the `ADPCMDecoder()` function, which is listed in **Appendix C: "Generic `ADPCMDecoder()` Function"**. This function requires three 16-bit variables and one 8-bit variable.

**TABLE 2: `ADPCMDecoder()` STEP-BY-STEP FUNCTIONS**

| | |
|---|---|
| 1. `ADPCMDecoder` takes an 8-bit number containing the 4-bit ADPCM code (0-15) and returns a 16-bit signed number (speech sample, 32767 to -32768). | `signed long ADPCMDecoder( char code )` |
| 2. Restore the previous values of predicted sample ($s_p$) and quantizer step size index. | `predsample = state.prevsample;`<br>`index = state.previndex;` |
| 3. Find the quantizer step size ($q$) from a table look up using the quantizer step size index. | `step = StepSizeTable[index];` |
| 4. Inverse quantize the ADPCM code ($t$) into a predicted difference ($d_q$) using the quantizer step size ($q$). | `diffq = step >> 3;`<br>`if(code & 4)`<br>`  diffq += step;`<br>`if(code & 2)`<br>`  diffq += step >> 1;`<br>`if(code & 1)`<br>`  diffq += step >> 2;` |
| 5. Fixed predictor computes new predicted sample ($s_r$) by adding the old predicted sample ($s_p$) to the predicted difference ($d_q$). | `if(code & 8)`<br>`  predsample -= diffq;`<br>`else`<br>`  predsample += diffq;` |
| 6. Check for overflow of the new predicted sample ($s_r$). $s_r$, which is a signed 16-bit sample, must be in the range of 32767 to -32768. | `if(predsample > 32767)`<br>`  predsample = 32767;`<br>`else if(predsample < -32768)`<br>`  predsample = -32768;` |
| 7. Find the new quantizer step size ($q$) by adding the previous index and a table look up using the ADPCM code ($t$). | `index += IndexTable[code];` |
| 8. Check for overflow of the new quantizer step size index. | `if(index < 0)`<br>`  index = 0;`<br>`if(index > 88)`<br>`  index = 88;` |
| 9. Save the new predicted sample ($s_r$) and quantizer step size index for next iteration. | `state.prevsample = predsample;`<br>`state.previndex = index;` |
| 10. Return the new sample ($s_r$). | `return ( predsample );` |

## IMA ADPCM REFERENCE ALGORITHM

The IMA, specifically the Digital Audio Technical Working Group, was a trade association with representatives from companies such as Compaq®, Apple® Computers, Crystal Semiconductor, DEC, Hewlett-Packard, Intel®, Microsoft®, Sony® and Texas Instruments™. This group was working towards a standard that defines the exchange of high quality audio data between computing platforms. The algorithm from Intel DVI (Digital Video Interactive) was selected as the standard due to its audio dynamic range and low data rate. The recommended digital audio exchange formats are given in Table 3.

The algorithms that are implemented in this application note were derived from the IMA ADPCM Reference Algorithm. The data format is 8.0 kHz, mono, 4-bit ADPCM. Essentially, the compression and decompression use an adaptive quantization with fixed prediction. The adaptive quantization is based on a table look up first developed by Intel DVI for the IMA. **Appendix D: "Interactive Multimedia Association Information"** remains to show the references as listed in the original application note published in 1997. The IMA is no longer an operational organization.

## PERFORMANCE

Through experimentation, it has been determined that a 16 MHz oscillator will provide enough overhead to execute an encode or decode routine at a sample rate of 8.0 kHz. This assumes that the on-chip A/D converter is used for encoding and the PWM module for decoding.

**TABLE 3: DIGITAL AUDIO EXCHANGE FORMATS**

| Sampling Rate | Mono/Stereo | Data Format | Notes |
|---|---|---|---|
| 8.0 kHz | mono | 8-bit m-Law PCM | CCITT G.711 Standard |
| | | 8-bit A-Law PCM | CCITT G.711 Standard |
| | | 4-bit ADPCM | DVI Algorithm |
| 11.025 kHz | mono/stereo | 8-bit Linear PCM | Macintosh® & MP-C Standard |
| | | 4-bit ADPCM | DVI Algorithm |
| 22.05 kHz | | 8-bit Linear PCM | Macintosh & MPC Standard |
| | | 4-bit ADPCM | DVI Algorithm |
| 44.10 kHz | | 16-bit Linear PCM | CD-DA Standard |
| | | 4-bit ADPCM | DVI Algorithm |

Table 4 illustrates the amount of program and data memory that the ADPCM algorithms consume for the various PIC microcontrollers. The program memory numbers may change depending on the specific device being used, and the C compiler and optimization levels. The table memory column shows how much program memory is used to store the two look-up tables used by the ADPCM algorithm.

The IMA ADPCM algorithm can be used on the PIC16 microcontrollers but is not shown in this application note.

**TABLE 4: DEVICE MEMORY CONSUMED**

| Device | | Program Memory (words) | Data Memory (bytes) | Table Look Up (words) |
|---|---|---|---|---|
| PIC18 | Encode | 736 | 13 | 97 |
| | Decode | 484 | 10 | |
| PIC24/ dsPIC® | Encode | 690 | 26 | 196 |
| | Decode | 276 | 20 | |

## APPLICATION

The hardware for this application note implements only the decompression algorithm. The compression algorithm, `ADPCMEncoder()`, can be added for applications that need this functionality. This application note uses the Speech Playback PICtail™ Plus Daughter Board (Part #AC164125) and a PICDEM™ HPC Explorer Board (DM183022) for PIC18 or an Explorer 16 Development Board (DM240001) for 16-bit MCUs.

This design shows how to implement a talking thermometer. The PIC18F67J10 microcontroller has four sections of code: read temperature using TC77 through the SPI module, convert binary temperature into ASCII, decode the individual speech elements and playback through a CCP module. The compressed speech data is stored in the internal program memory of the PIC18F67J10.

Prerecorded speech segments for the numbers, 1-20, 30, 40, 50, 60, 70, 80, 90 and 100, and the words, celsius, fahrenheit and degrees, are compressed and then stored in program memory. The individual speech files are organized into a single file using the MPFS file system that is used for the Microchip TCP/IP stack. See *AN833 "Microchip TCP/IP Stack Application Note"* (DS00833) for details on the MPFS file system.

Speech regeneration is accomplished by using a Capture/Compare/PWM (CCP) module of the PIC18F67J10. The PWM module is configured for a period of 16 kHz. This allows each sample of the speech signal to be output for two PWM periods for an oscillator frequency of 32 MHz. The period is calculated by using the following equation from **Section 16.4.1 "PWM Period"** of the PIC18F87J10 Family Data Sheet (DS39663). From the following calculation, PR2 is set to a value of 249.

**EQUATION 1:**

$$\begin{aligned} \text{PWM Period} &= [\text{PR2} + 1] \cdot 4 \cdot \text{TOSC} \cdot (\text{TMR2 Prescale Value}) \\ 1/16\text{ kHz} &= [\text{PR2} + 1] \cdot 4 \cdot (1 / 32\text{ MHz}) \cdot 2 \\ 62.5\text{ ms} &= [\text{PR2} + 1] \cdot 4 \cdot 31.25\text{ ns} \cdot 2 \\ 250 &= \text{PR2} + 1 \\ 249 &= \text{PR2} \end{aligned}$$

The CCP module has the capability of up to 10-bit resolution for the PWM duty cycle. The maximum resolution of the duty cycle that can be achieved for a 16 kHz period can be calculated using the following equation from **Section 16.4.2 "PWM Duty Cycle"** of the PIC18F87J10 Family Data Sheet.

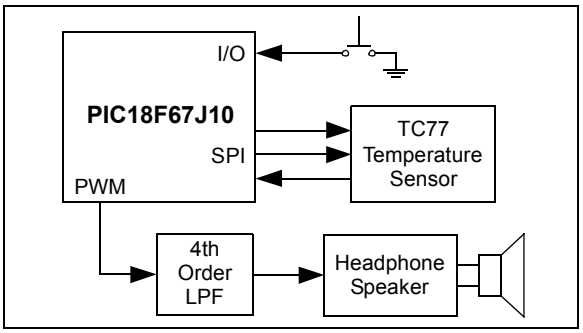
**EQUATION 2:**

$$\begin{aligned} \text{PWM Duty Cycle} &= \text{CCPRxL:CCPxCON<5:4>} \cdot \text{Tosc} \cdot (\text{TMR2 Prescale Value}) \\ \text{where:} & \\ \text{CCPRxL:CCPxCON<5:4>} &= 2^x \\ \text{where:} & \\ x &= \text{bits of resolution} \\ 1/16\text{ kHz} &= 2^x \cdot (1 / 32\text{ MHz}) \cdot 2 \\ 62.5\text{ ms} &= 2^x \cdot 31.25\text{ ns} \cdot 2 \\ 1000 &= 2^x \\ \log(1000) &= \log(2^x) \\ \log(1000) &= x \cdot \log(2) \\ 9.96 &= x \end{aligned}$$

A PWM duty cycle with close to 10 bits of resolution may be used with a period of 16 kHz. The upper 9 bits of each speech sample are used to set the PWM duty cycle (CCPR1L = sample<15:9>, CCP1CON<5:4> = sample<8:7>). This PWM speech signal is passed through a 4th order Butterworth low-pass filter with a corner frequency of 4 kHz. The low-pass filter converts the PWM into an analog voltage level. Finally, the analog voltage level is amplified by a National Semiconductor LM4853 before entering the speaker.

Every 125 μs (8 kHz), one ADPCM code must be converted into a speech sample for output to the PWM. This frame of 125 μs relates to 1000 instruction cycles for an internal 8 MHz oscillator with PLL enabled for a frequency of 32 MHz (125 nS). The ADPCM decode routine, Flash program memory reads, conversion of binary to ASCII and writes to the PWM must be accomplished within 1000 instruction cycles.

**FIGURE 3: APPLICATION HARDWARE BLOCK DIAGRAM**



To implement audio recording, the following changes to the application hardware would have to be made:

- Internal Flash program memory may not be fast enough to record at 8 kHz. Therefore, an external memory, such as the SST25LF020A 2 Mb Serial Flash from SST, could be used. This device can program a byte of data in 14 μs.
- Add a microphone with input filtering.
- If needed, use the on-board A/D converter of the PIC18F67J10 device for 10-bit recording; or
- Use an external 12- to 16-bit A/D converter, such as the Microchip MCP3221.

The following is the sequence of events to record and store speech data. A timer would be used to set the sample rate. When the timer overflowed, an A/D conversion would be started.

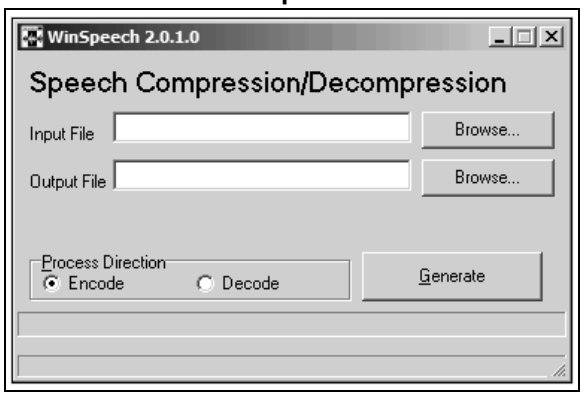
1. Start an A/D conversion when the timer overflows.
2. Read the sample and call the routine, `ADPCMEncoder()`.
3. Store the result in the upper 4 bits of a temporary variable.
4. Start an A/D conversion when the timer overflows.
5. Read the sample and call the routine, `ADPCMEncoder()`.
6. Store the result in the lower 4 bits of the temporary variable.
7. Write the temporary variable to the nonvolatile memory.
8. Go to step 1.

Typical conversion time for the PIC18F67J10 device is 25 μs. Assuming that the oscillator frequency is 32 MHz and the sample rate is 8 kHz, the encoder routine takes approximately 38 μs (~300 instruction cycles x 125 ns) to complete. This leaves approximately 62 μs or ~500 instruction cycles to read the A/D converter, write to the nonvolatile memory and complete any other tasks. An external A/D converter that is faster could be used to increase the amount of time for processing other tasks.

## COMPUTER PROGRAM

WinSpeech was developed to encode and decode speech files. The encode operation takes a 16-bit unsigned raw input file and creates a compressed speech output file. Each byte in this file contains two 4-bit ADPCM codes, where the upper byte is decoded first, followed by the lower byte. Decoding is the opposite of taking 4-bit ADPCM codes and saving the speech in unsigned 16-bit raw format. Figure 4 shows the graphical user interface of WinSpeech. Refer to **Appendix A: "Source Code"** for information on obtaining the WinSpeech executable and source code referenced in this application note.

**FIGURE 4: WinSpeech PC PROGRAM**




WinSpeech uses the same basic routines that can be found in the zip archive (see **Appendix A: "Source Code"**). Speech should be recorded at the highest possible sample rate, normally 44.1. The sound editing program, such as the GoldWave™ Digital Audio Editor (see **"References"**), should be used to edit the file down to 8 kHz or whatever the playback rate is for the application. The resulting sound file should be saved in 16-bit mono raw format.

A raw sound file is recorded on the PC using a 16-bit sound card and PC program, such as GoldWave. Speech is recorded at 8.0 kHz in 16-Bit Mono mode. This raw speech file is processed by the encoder part of WinSpeech, which creates a file with the ADPCM values. This file can now be burned into nonvolatile memory for use with the application hardware. The decoder part of WinSpeech can take this file and create a new 16-bit raw speech file. The sound editing program and sound card on the PC can play this file to ensure that the ADPCM compression/decompression routines are working properly.

## CONCLUSION

The final results of the application hardware using the PIC18F67J10 are:

- Decompression only:
  - 484 bytes out of 128 Kbytes of program memory
  - 20 bytes out of 3936 bytes of data memory
- Hardware used:
  - CCP1 configured for PWM (9-bit duty cycle, 32 kHz period speech output)
  - Timer2 to set the 8.0 kHz output sample rate
  - 59527 bytes of program memory to store compressed voice data files

## REFERENCES


1. *Recommended Practices for Enhancing Digital Audio Compatibility in Multimedia Systems*, Revision 3.00, Interactive Multimedia Association, October 21, 1992.
2. *Digital Audio Special Edition Proceedings*, Volume 2, Issue 2, Interactive Multimedia Association, May 1992.
3. Panos E. Papamichalis Ph.D., *Practical Approaches to Speech Coding*, Prentice-Hall Inc., Englewood Cliffs, N.J, 1987.
4. Adaptive Differential Pulse Code Modulation, *Digital Signal Processing Applications using the ADSP-2100 Family*, Volume 1, Analog Devices, Prentice-Hall, Englewood Cliffs, N.J., 1992.
5. 32-kbits/s ADPCM with the TMS32010, *Digital Signal Processing Applications with the TMS320 Family*, SPRA012, Jay Reimer, Mike McMahan and Masud Arjmand, Texas Instruments, 1986.
6. GoldWave Speech Processing Program, Goldwave, Inc., 2006. URL: www.goldwave.com.

## APPENDIX A: SOURCE CODE

All of the software covered in this application note is available as a single WinZip archive file. The archive may be downloaded from the Microchip corporate Web site at:

**www.microchip.com**

## APPENDIX B: GENERIC `ADPCMEncoder()` FUNCTION

```
/* Table of index changes */
const int IndexTable[16] = {
    0xff, 0xff, 0xff, 0xff, 2, 4, 6, 8,
    0xff, 0xff, 0xff, 0xff, 2, 4, 6, 8
};

/* Quantizer step size lookup table */
const long StepSizeTable[89] = {
    7, 8, 9, 10, 11, 12, 13, 14, 16, 17,
    19, 21, 23, 25, 28, 31, 34, 37, 41, 45,
    50, 55, 60, 66, 73, 80, 88, 97, 107, 118,
    130, 143, 157, 173, 190, 209, 230, 253, 279, 307,
    337, 371, 408, 449, 494, 544, 598, 658, 724, 796,
    876, 963, 1060, 1166, 1282, 1411, 1552, 1707, 1878, 2066,
    2272, 2499, 2749, 3024, 3327, 3660, 4026, 4428, 4871, 5358,
    5894, 6484, 7132, 7845, 8630, 9493, 10442, 11487, 12635, 13899,
    15289, 16818, 18500, 20350, 22385, 24623, 27086, 29794, 32767
};

signed long diff;               /* Difference between sample and predicted sample */
long step;                      /* Quantizer step size */
signed long predsample;         /* Output of ADPCM predictor */
signed long diffq;              /* Dequantized predicted difference */
int index;                      /* Index into step size table */

/*****************************************************************************
*       ADPCMEncoder - ADPCM encoder routine                                 *
*****************************************************************************
*       Input Variables:                                                     *
*               signed long sample - 16-bit signed speech sample             *
*       Return Variable:                                                     *
*               char - 8-bit number containing the 4-bit ADPCM code          *
*****************************************************************************/
char ADPCMEncoder( signed long sample )
{
    int code;               /* ADPCM output value */
    int tempstep;           /* Temporary step size */

    /* Restore previous values of predicted sample and quantizer step
       size index
    */
    predsample = state.prevsample;
    index = state.previndex;
    step = StepSizeTable[index];


    /* Compute the difference between the actual sample (sample) and the
       the predicted sample (predsample)
    */
    diff = sample - predsample;
    if(diff >= 0)
        code = 0;
    else
    {
        code = 8;
        diff = -diff;
    }

    /* Quantize the difference into the 4-bit ADPCM code using the
       the quantizer step size
    */
    tempstep = step;
    if( diff >= tempstep )
```

```
    {
        code |= 4;
        diff -= tempstep;
    }
    tempstep >>= 1;
    if( diff >= tempstep )
    {
        code |= 2;
        diff -= tempstep;
    }
    tempstep >>= 1;
    if( diff >= tempstep )
        code |= 1;

    /* Inverse quantize the ADPCM code into a predicted difference
       using the quantizer step size
    */
    diffq = step >> 3;
    if( code & 4 )
        diffq += step;
    if( code & 2 )
        diffq += step >> 1;
    if( code & 1 )
        diffq += step >> 2;

    /* Fixed predictor computes new predicted sample by adding the
       old predicted sample to predicted difference
    */
    if( code & 8 )
        predsample -= diffq;
    else
        predsample += diffq;

    /* Check for overflow of the new predicted sample
    */
    if( predsample > 32767 )
        predsample = 32767;
    else if( predsample < -32768 )
        predsample = -32768;

    /* Find new quantizer stepsize index by adding the old index
       to a table lookup using the ADPCM code
    */
    index += IndexTable[code];

    /* Check for overflow of the new quantizer step size index
    */
    if( index < 0 )
        index = 0;
    if( index > 88 )
        index = 88;

    /* Save the predicted sample and quantizer step size index for
       next iteration
    */
    state.prevsample = predsample;
    state.previndex = index;

    /* Return the new ADPCM code */
    return ( code & 0x0f );
}
```

## APPENDIX C: GENERIC `ADPCMDecoder()` FUNCTION

```
/* Table of index changes */
const int IndexTable[16] = {
    0xff, 0xff, 0xff, 0xff, 2, 4, 6, 8,
    0xff, 0xff, 0xff, 0xff, 2, 4, 6, 8
};

/* Quantizer step size lookup table */
const long StepSizeTable[89] = {
    7, 8, 9, 10, 11, 12, 13, 14, 16, 17,
    19, 21, 23, 25, 28, 31, 34, 37, 41, 45,
    50, 55, 60, 66, 73, 80, 88, 97, 107, 118,
    130, 143, 157, 173, 190, 209, 230, 253, 279, 307,
    337, 371, 408, 449, 494, 544, 598, 658, 724, 796,
    876, 963, 1060, 1166, 1282, 1411, 1552, 1707, 1878, 2066,
    2272, 2499, 2749, 3024, 3327, 3660, 4026, 4428, 4871, 5358,
    5894, 6484, 7132, 7845, 8630, 9493, 10442, 11487, 12635, 13899,
    15289, 16818, 18500, 20350, 22385, 24623, 27086, 29794, 32767
};

long step;                      /* Quantizer step size */
signed long predsample;         /* Output of ADPCM predictor */
signed long diffq;              /* Dequantized predicted difference */
int index;                      /* Index into step size table */

/*******************************************************************************
*       ADPCMDecoder - ADPCM decoder routine                                   *
*******************************************************************************
*       Input Variables:                                                       *
*               char code - 8-bit number containing the 4-bit ADPCM code       *
*       Return Variable:                                                       *
*               signed long - 16-bit signed speech sample                      *
*******************************************************************************/
signed long ADPCMDecoder(char code )
{

    /* Restore previous values of predicted sample and quantizer step
       size index
    */
    predsample = state.prevsample;
    index = state.previndex;

    /* Find quantizer step size from lookup table using index
    */
    step = StepSizeTable[index];

    /* Inverse quantize the ADPCM code into a difference using the
       quantizer step size
    */
    diffq = step >> 3;
    if( code & 4 )
        diffq += step;
    if( code & 2 )
        diffq += step >> 1;
    if( code & 1 )
        diffq += step >> 2;

    /* Add the difference to the predicted sample
    */
    if( code & 8 )
        predsample -= diffq;
    else
        predsample += diffq;
```

```
    /* Check for overflow of the new predicted sample
    */
    if( predsample > 32767 )
        predsample = 32767;
    else if( predsample < -32768 )
        predsample = -32768;

    /* Find new quantizer step size by adding the old index and a
       table lookup using the ADPCM code
    */
    index += IndexTable[code];

    /* Check for overflow of the new quantizer step size index
    */
    if( index < 0 )
        index = 0;
    if( index > 88 )
        index = 88;


    /* Save predicted sample and quantizer step size index for next
       iteration
    */
    state.prevsample = predsample;
    state.previndex = index;

    /* Return the new speech sample */
    return( predsample );

}
```

## APPENDIX D: INTERACTIVE MULTIMEDIA ASSOCIATION INFORMATION

**Note:** The IMA is no longer a functioning organization. **Appendix D: "Interactive Multimedia Association Information"** is retained as is from the original publication of this application note (AN643) in 1997. The Digital Audio Doc-Pac is no longer published, and the web site, www.ima.org, has been reassigned to a new company.

The IMA ADPCM Reference Algorithm is contained in the Digital Audio Doc-Pac. The Doc-Pac contains the following information:

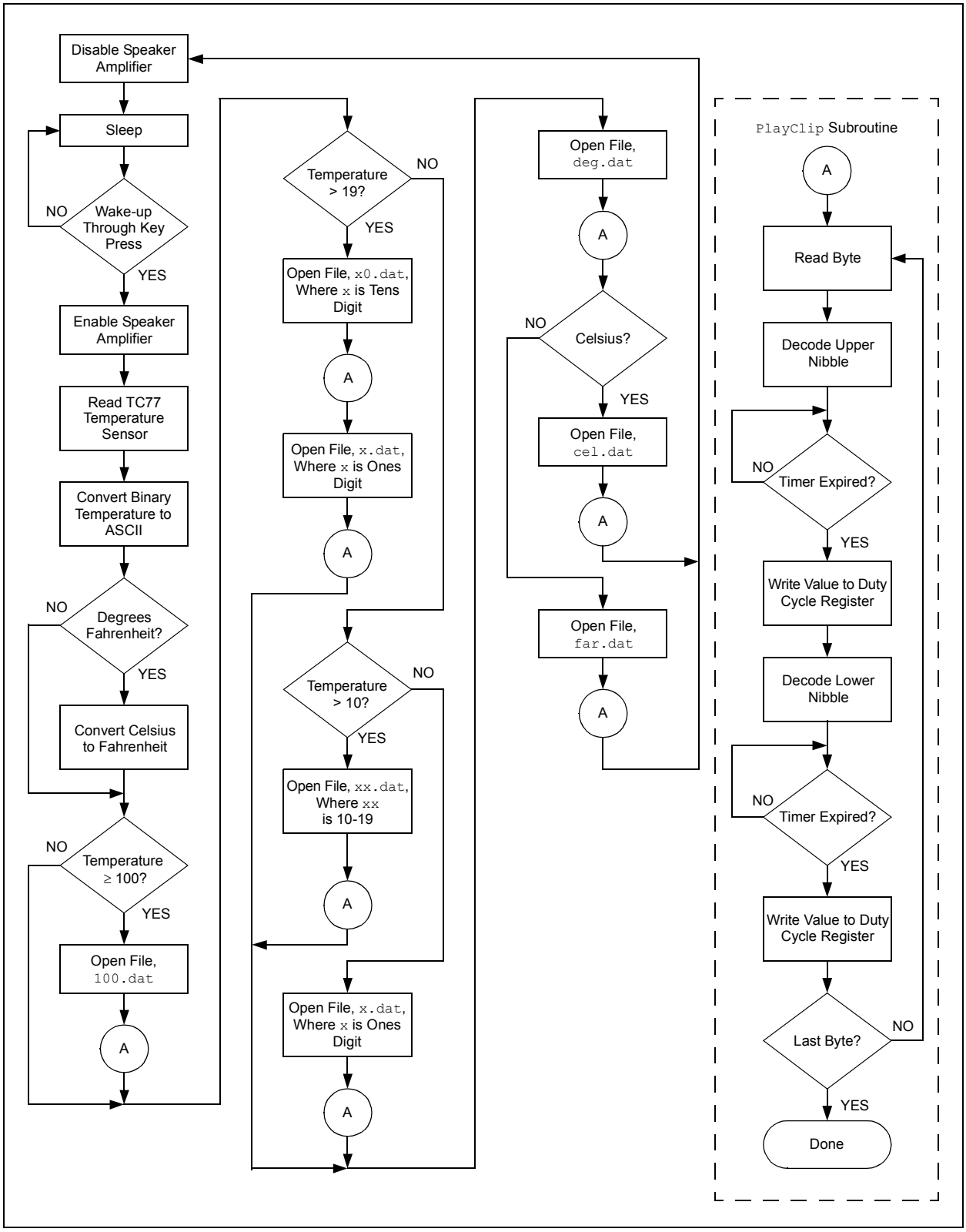
- *About IMA Digital Audio* – describes the IMA's activities in digital audio.
- IMA *Recommended Practices for Enhancing Digital Audio Compatibility in Multimedia Systems* (version 3.0) – this document contains the official technical recommendations including the ADPCM algorithm.
- *IMA Digital Audio Special Edition Proceedings* – this document contains detailed notes of the evaluation process, code fragments and testing methodologies.
- Floppy disk with code fragments.
- Contact information for companies supporting IMA ADPCM.

The IMA can be reached at:

Interactive Multimedia Association
48 Maryland Avenue, Suite 202
Annapolis, MD 21401-8011 USA
Tel: (410)-626-1380
Fax: (410)-263-0590
http://www.ima.org

## APPENDIX E: FLOWCHART

### FIGURE E-1: APPLICATION FIRMWARE FLOWCHART



**NOTES:**

**Note the following details of the code protection feature on Microchip devices:**

- Microchip products meet the specification contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is one of the most secure families of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the Microchip products in a manner outside the operating specifications contained in Microchip's Data Sheets. Most likely, the person doing so is engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as "unbreakable."

Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our products. Attempts to break Microchip's code protection feature may be a violation of the Digital Millennium Copyright Act. If such acts allow unauthorized access to your software or other copyrighted work, you may have a right to sue for relief under that Act.



**Trademarks**



QUALITY MANAGEMENT SYSTEM
CERTIFIED BY DNV
ISO/TS 16949:2002

*Microchip received ISO/TS-16949:2002 certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona; Gresham, Oregon and design centers in California and India. The Company's quality system processes and procedures are for its PIC® MCUs and dsPIC® DSCs, KEELOQ® code hopping devices, Serial EEPROMs, microperipherals, nonvolatile memory and analog products. In addition, Microchip's quality system for the design and manufacture of development systems is ISO 9001:2000 certified.*

# WORLDWIDE SALES AND SERVICE

## AMERICAS

**Corporate Office**
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7200
Fax: 480-792-7277
Technical Support:
http://support.microchip.com
Web Address:
www.microchip.com

**Atlanta**
Duluth, GA
Tel: 678-957-9614
Fax: 678-957-1455

**Boston**
Westborough, MA
Tel: 774-760-0087
Fax: 774-760-0088

**Chicago**
Itasca, IL
Tel: 630-285-0071
Fax: 630-285-0075

**Dallas**
Addison, TX
Tel: 972-818-7423
Fax: 972-818-2924

**Detroit**
Farmington Hills, MI
Tel: 248-538-2250
Fax: 248-538-2260

**Kokomo**
Kokomo, IN
Tel: 765-864-8360
Fax: 765-864-8387

**Los Angeles**
Mission Viejo, CA
Tel: 949-462-9523
Fax: 949-462-9608

**Santa Clara**
Santa Clara, CA
Tel: 408-961-6444
Fax: 408-961-6445

**Toronto**
Mississauga, Ontario,
Canada
Tel: 905-673-0699
Fax: 905-673-6509

## ASIA/PACIFIC

**Asia Pacific Office**
Suites 3707-14, 37th Floor
Tower 6, The Gateway
Harbour City, Kowloon
Hong Kong
Tel: 852-2401-1200
Fax: 852-2401-3431

**Australia - Sydney**
Tel: 61-2-9868-6733
Fax: 61-2-9868-6755

**China - Beijing**
Tel: 86-10-8528-2100
Fax: 86-10-8528-2104

**China - Chengdu**
Tel: 86-28-8665-5511
Fax: 86-28-8665-7889

**China - Fuzhou**
Tel: 86-591-8750-3506
Fax: 86-591-8750-3521

**China - Hong Kong SAR**
Tel: 852-2401-1200
Fax: 852-2401-3431

**China - Qingdao**
Tel: 86-532-8502-7355
Fax: 86-532-8502-7205

**China - Shanghai**
Tel: 86-21-5407-5533
Fax: 86-21-5407-5066

**China - Shenyang**
Tel: 86-24-2334-2829
Fax: 86-24-2334-2393

**China - Shenzhen**
Tel: 86-755-8203-2660
Fax: 86-755-8203-1760

**China - Shunde**
Tel: 86-757-2839-5507
Fax: 86-757-2839-5571

**China - Wuhan**
Tel: 86-27-5980-5300
Fax: 86-27-5980-5118

**China - Xian**
Tel: 86-29-8833-7252
Fax: 86-29-8833-7256

## ASIA/PACIFIC

**India - Bangalore**
Tel: 91-80-4182-8400
Fax: 91-80-4182-8422

**India - New Delhi**
Tel: 91-11-4160-8631
Fax: 91-11-4160-8632

**India - Pune**
Tel: 91-20-2566-1512
Fax: 91-20-2566-1513

**Japan - Yokohama**
Tel: 81-45-471- 6166
Fax: 81-45-471-6122

**Korea - Daegu**
Tel: 82-53-744-4301
Fax: 82-53-744-4302

**Korea - Seoul**
Tel: 82-2-554-7200
Fax: 82-2-558-5932 or
82-2-558-5934

**Malaysia - Penang**
Tel: 60-4-646-8870
Fax: 60-4-646-5086

**Philippines - Manila**
Tel: 63-2-634-9065
Fax: 63-2-634-9069

**Singapore**
Tel: 65-6334-8870
Fax: 65-6334-8850

**Taiwan - Hsin Chu**
Tel: 886-3-572-9526
Fax: 886-3-572-6459

**Taiwan - Kaohsiung**
Tel: 886-7-536-4818
Fax: 886-7-536-4803

**Taiwan - Taipei**
Tel: 886-2-2500-6610
Fax: 886-2-2508-0102

**Thailand - Bangkok**
Tel: 66-2-694-1351
Fax: 66-2-694-1350

## EUROPE

**Austria - Wels**
Tel: 43-7242-2244-39
Fax: 43-7242-2244-393

**Denmark - Copenhagen**
Tel: 45-4450-2828
Fax: 45-4485-2829

**France - Paris**
Tel: 33-1-69-53-63-20
Fax: 33-1-69-30-90-79

**Germany - Munich**
Tel: 49-89-627-144-0
Fax: 49-89-627-144-44

**Italy - Milan**
Tel: 39-0331-742611
Fax: 39-0331-466781

**Netherlands - Drunen**
Tel: 31-416-690399
Fax: 31-416-690340

**Spain - Madrid**
Tel: 34-91-708-08-90
Fax: 34-91-708-08-91

**UK - Wokingham**
Tel: 44-118-921-5869
Fax: 44-118-921-5820

06/25/07